Avalanche: An Event Dispatcher for SNO+

(as in, a lot of sno(+), very fast)

Andy Mastbaum*
The University of Pennsylvania

November 14, 2011

1 Introduction

The "dispatcher" is the server software responsible for providing a stream of built events to various client software. Currently-planned client software includes the event viewer and detector monitoring tools. The dispatcher requires:

High Throughput The dispatcher must be able to keep up with the maximum data rate of 450 Mbps, simultaneously to all clients.

Scalability The dispatcher should support a large number of concurrent client connections without impact on performance.

Compatibility The dispatched data stream must be reasonably language-agnostic, since the language of the client software is unknown.

An obvious platform choice would be ROOT's network I/O classes (TPSocket, etc.). However, the next-generation socket library ZeroMQ[1] provides a superior alternative. ZeroMQ is naturally asynchronous, natively supports fan-out, and implementations are typically faster than UNIX sockets. ZeroMQ's fan-out allows clients to "subscribe" to a stream, optionally applying a filter (e.g. only events, no run headers). Bindings exist for all major languages.

2 Data Format

It has been decided that the "packed" events coming from the event builder will be in a ROOT format, with a structure developed by G. D. Orebi Gann and J. Kaspar. The packed format preserves the "raw" data structures from the detector front-end: for example, charge and time data is stored in 96-bit PMT bundles. This is in contrast the 'full' or 'unpacked' data structure used in RAT, where values are broken into separate, histogrammable fields.

The packed data is essentially a stream of TObjects, which may be event data, event-level headers, or run-level headers. For convenience of storage these are written to a `TTree`. Since `TTree`s are homogeneous lists, various data types must inherit from the same generic superclass to be stored in the tree. The data type must also be stored, so that objects can be properly re-cast when read back. The packed format `TTree` stores `RAT::DS::PackedRec` objects, which in turn contain a single `GenericRec` and an integer `RecordType`. All data types (events, headers) inherit from a class `GenericRec`, which is empty and exists only to support this polymorphism.

A full listing of packed format classes and their members is given in the appendix.

*mastbaum@hep.upenn.edu
[1] http://www.zeromq.org

3 Network Layer

The avalanche server broadcasts `PackedRec` objects, serialized using a ROOT `TBufferFile`. Client software must create a `TBufferFile` using the packet data, read the `PackedRec` object out of it, and cast the `PackedRec`'s `Rec` object to the correct type based on the value of `RecordType`. Examples in C++ and Python are included.

4 Dispatching Simulated RAT Events

`avalanche` includes a RAT output processor which broadcasts packed events on ZeroMQ socket. Clients may thus listen to a RAT-simulated data stream for testing and development. The processor, DispatchEvents, is provided in the `rat` subfolder of the avalanche distribution.

5 Dispatching Detector Events

A C++ library with a very simple API is provided in the `lib` subfolder.

A server consists of a single `AvalancheServer` object, constructed with a given socket address. The `AvalancheServer` has one method, `sendObject`, which dispatches a ROOT `TObject`. For example, to send a histogram `TH1F* h1`:

```
AvalancheServer* serv = new AvalancheServer("tcp://localhost:5024");
serv->sendObject(h1);
```

Any number of clients may be subscribed to the stream on port 5024.

6 Writing Dispatcher Clients

Client software subscribes to a ZeroMQ TCP socket and deserializes ROOT `TObjects`. The former is achieved with the ZeroMQ API, available in all major languages (and many minor ones). The latter imposes some language restrictions, as ROOT bindings exist for a limited number. C++, Python (PyROOT), and Ruby (RubyROOT) are the best-supported choices.

6.1 C++

In C++, ZeroMQ sockets are handled by the `zmq` library, and deserialization with normal ROOT classes.

Consider the following example (available in `examples/client_cpp`):

```
#include <zmq.hpp>
#include <time.h>
#include <iostream>

#include <RAT/DS/Root.hh>
#include <RAT/DS/PackedEvent.hh>
#include <TBuffer.h>
#include <TBufferFile.h>

int main(int argc, char *argv[])
{
    // make a zeromq socket
    zmq::context_t context(1);
    zmq::socket_t subscriber(context, ZMQ_SUB);
    subscriber.setsockopt(ZMQ_SUBSCRIBE, "", 0); //filter, strlen (filter));
    subscriber.bind("tcp://*:5024");
```

```
    while (1) {
        // listen for incoming messages
        zmq::message_t message;
        subscriber.recv(&message);

        // read message data into a TBufferFile and deserialize
        TBufferFile buf(TBuffer::kRead, message.size(), message.data(), false);
        RAT::DS::PackedRec* rec = \
          (RAT::DS::PackedRec*)(buf.ReadObjectAny(RAT::DS::PackedRec::Class()));
        if (rec)
            std::cout << "Received PackedRec of type " << rec->RecordType << std::endl;
        else
            std::cout << "Error deserializing message data" << std::endl;
        delete rec;
    }

    return 0;
}
```

First, we set up a ZMQ socket to subscribe to a server. Then, we wait to receive packets. Packet data comes as `zmq::message_t` objects, which have an `int zmq::message_t::size()` and a `void* zmq::message_t::data()`. Using the size and data to construct a `TBufferFile`, we may read out the deserialized `PackedRec` with `TBufferFile::ReadObject` or `TBufferFile::ReadObjectAny`.

In this example, we just print the record type. A real client would at this point cast the member `Rec` object to the correct type, extract data, and perform some operations.

6.2 Python

In Python, ZeroMQ sockets are handled by the zmq module, available from github (zeromq/pyzmq) or via `easy_install`. Deserialization is done with ROOT classes accessed through PyROOT.

Consider the following example (available in `examples/client_python`):

```
import sys
import array
import zmq
from rat import ROOT

if __name__ == '__main__':
    if len(sys.argv) > 1:
        address = sys.argv[1]
    else:
        address = 'tcp://*:5024'

    # set up zeromq socket
    context = zmq.Context()
    socket = context.socket(zmq.SUB)
    socket.setsockopt(zmq.SUBSCRIBE, '')
    socket.bind(address)

    while True:
        msg = socket.recv(copy=False)
        # buffer contains null characters, so wrap in array to pass to c
        b = array.array('c', msg.bytes)
        buf = ROOT.TBufferFile(ROOT.TBuffer.kRead, len(b), b, False, 0)
```

```
rec = buf.ReadObject(ROOT.RAT.DS.PackedRec.Class())

if rec:
    print 'Received PackedRec of type', rec.RecordType
else:
    print 'Error deserializing message data'
```

First, we set up a ZMQ socket to subscribe to a server. Then, we wait to receive packets. With `copy=False` in `recv()`, packet data comes as `zmq.core.message.Message` objects, which have a `buffer` (buffer) and a `bytes` (str) object.

To convert unterminated Python strings to null-terminated C strings, the Python/C interface reads a string only until it hits the first zero byte. To work around this, the buffer's byte array must be stored in a character `array.array` as shown. PyROOT correctly casts the `array` to a void pointer for the `TBufferFile` constructor.

Using the length of the character array and array itself to construct a `TBufferFile`, we may read out the deserialized `PackedRec` with `TBufferFile::ReadObject` or `TBufferFile::ReadObjectAny`.

In this example, we just print the record type. A real client would at this point cast the member `Rec` object to the correct type, extract data, and perform some operations.

6.3 Other Languages

Many other languages are supported by ZeroMQ, and at least Ruby has supported ROOT bindings. Implementations will be similar to those given in C++ and Python. For examples of ZeroMQ in your language, see https://github.com/imatix/zguide/tree/master/examples.

7 Appendix: Packed Data Model

This is a complete listing of all packed format classes, taken from RAT's PackedEvent.hh.

7.1 class GenericRec : public TObject

(empty)

7.2 class PackedRec : public TObject

- UInt_t RecordType
- GenericRec *Rec

7.2.1 Record Types

0. Empty
1. Detector event
2. RHDR
3. CAAC
4. CAST
5. TRIG
6. EPED

7.3 class PackedEvent : public GenericRec

- UInt_t MTCInfo[kNheaders] (6 words for the event header from the MTC)
- UInt_t RunID
- UInt_t SubRunID
- UInt_t NHits
- UInt_t EVOrder
- ULong64_t RunMask
- char PackVer
- char MCFlag
- char DataType
- char ClockStat10
- std::vector<PMTBundle> PMTBundles

7.4 class PMTBundle

- UInt_t Word[3]

7.5 class EPED : public GenericRec

- UInt_t GTDelayCoarse
- UInt_t GTDelayFine
- UInt_t QPedAmp
- UInt_t QPedWidth
- UInt_t PatternID
- UInt_t CalType
- UInt_t EventID (GTID of first events in this bank validity)
- UInt_t RunID (doublecheck on the run)

7.6 class TRIG : public GenericRec

- UInt_t TrigMask
- UShort_t Threshold[10]
- UShort_t TrigZeroOffset[10]
- UInt_t PulserRate
- UInt_t MTC_CSR
- UInt_t LockoutWidth
- UInt_t PrescaleFreq
- UInt_t EventID (GTID of first events in this banks validity)
- UInt_t RunID (doublecheck on the run)

7.6.1 Array Indices

Arrays correspond to:

0. N100Lo
1. N100Med
2. N100Hi
3. N20
4. N20LB
5. ESUMLo
6. ESUMHi
7. OWLn
8. OWLELo
9. OWLEHi

7.7 class RHDR : public GenericRec

- UInt_t Date
- UInt_t Time
- char DAQVer
- UInt_t CalibTrialID
- UInt_t SrcMask
- UInt_t RunMask
- UInt_t CrateMask
- UInt_t FirstEventID
- UInt_t ValidEventID
- UInt_t RunID (doublecheck on the run)

7.8 class CAST : public GenericRec

- UShort_t SourceID
- UShort_t SourceStat
- UShort_t NRopes
- float ManipPos[3]
- float ManipDest[3]
- float SrcPosUncert1
- float SrcPosUncert2[3]
- float LBallOrient
- std::vector<int> RopeID
- std::vector<float> RopeLen
- std::vector<float> RopeTargLen
- std::vector<float> RopeVel
- std::vector<float> RopeTens
- std::vector<float> RopeErr

7.9 class CAAC : public GenericRec

- float AVPos[3]
- float AVRoll[3] (roll, pitch and yaw)
- float AVRopeLength[7]